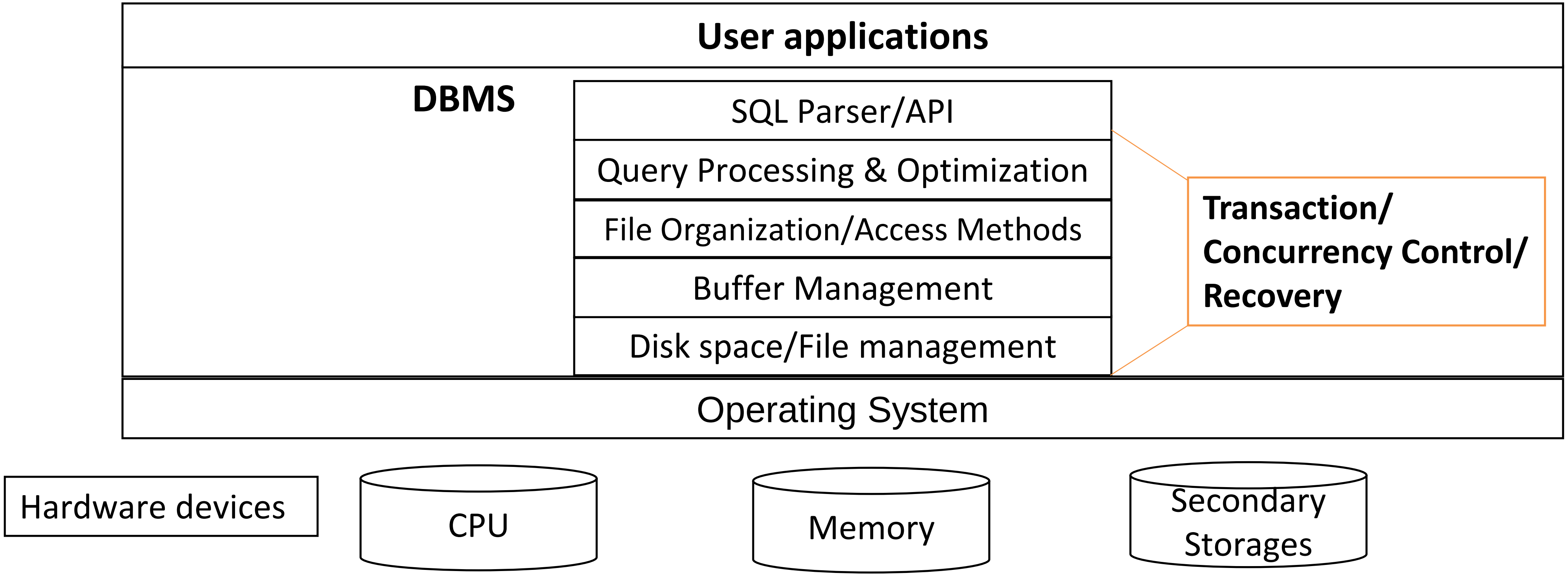
CSE462/562: Database Systems (Spring 22)

Lecture 17: Transactions

4/26/2022

University at Buffalo
Department of Computer Science and Engineering
School of Engineering and Applied Sciences

Big picture



User applications
DBMS
SQL Parser/API
Query Processing & Optimization
File Organization/Access Methods
Buffer Management
Disk space/File management
Transaction/
Concurrency Control/
Recovery
Operating System
Hardware devices
CPU
Memory
Secondary
Storages

What is a transaction?

Transaction:

```
BEGIN;
INSERT INTO A VALUES (…)
SELECT * from A;
DELETE FROM A WHERE …;
COMMIT;
```

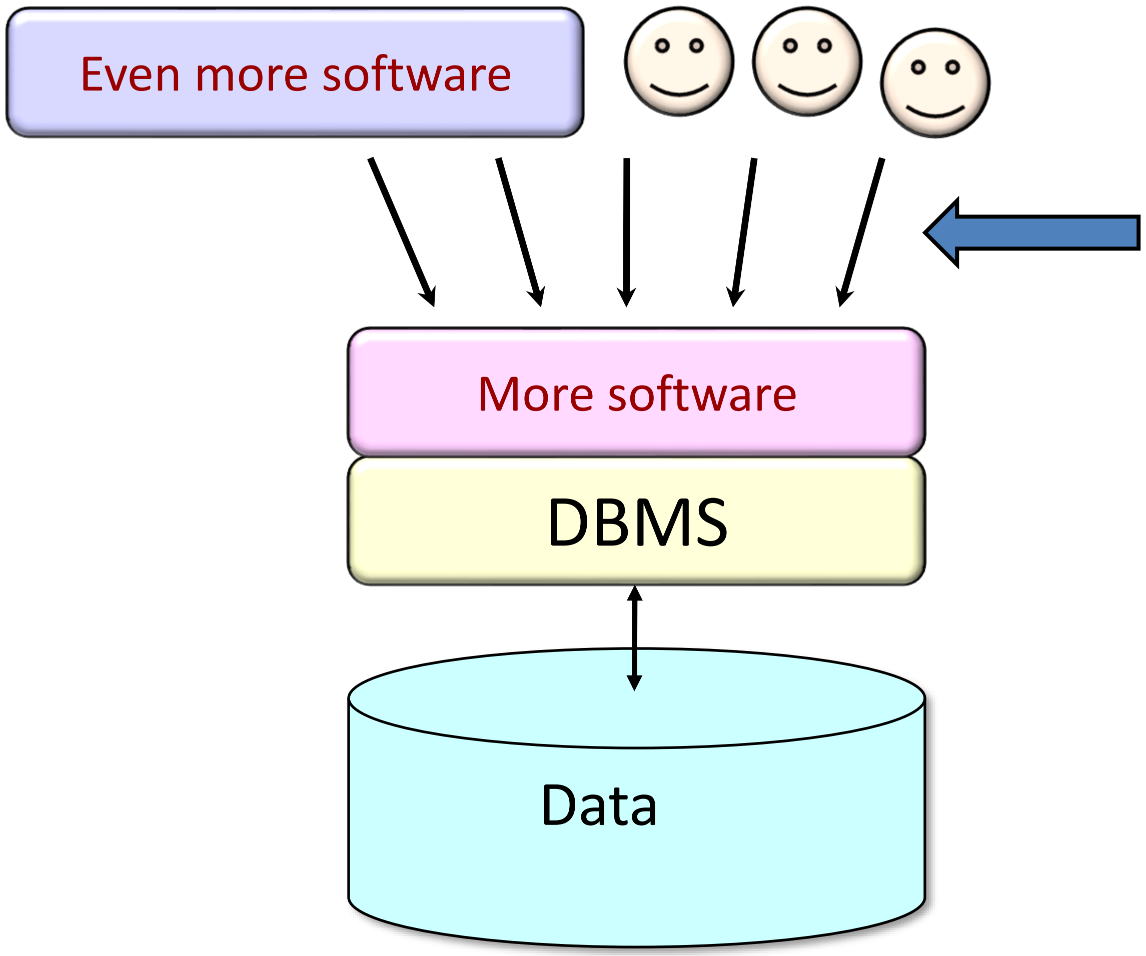
- A transaction is a sequence of one or more SQL operations treated as a unit
  - START/BEGIN [TRANSACTION] to start a new transaction
  - COMMIT: make all the changes by the current transaction permanent and visible
  - ROLLBACK/ABORT: revert all the changes by the current transaction

  - *Autocommit* turns each statement into a transaction
    - often enabled by default

Two independent motivation for transactions

- Concurrent database access
- Resilience to system failures

Motivation 1: concurrent Database Access



Select…
Update…
Create Table…
Drop Index…
Help…
Delete…

Concurrent access: attribute-level inconsistency

```
Update Account Set balance = balance + 1000
Where month(birthday) = 4
```

concurrent with …

```
Update Account Set balance = balance - 500
Where month(birthday) = 4
```

Actions involved: Get, Modify, Put. They may be interleaved!

Account(<u>acctno</u>, birthday, balance)
Sales(<u>saleid</u>, sale_date, acctno, amount, status)

Concurrent access: tuple-level inconsistency

```
Update Sales Set status = 'processing' Where saleid = 87654321
```

concurrent with …

```
Update Sales Set amt  = amt * 0.8 Where saleid = 87654321
```

Actions involved: Get, Modify, Put. They may be interleaved! Maybe only one of changes survives in the end.

```
Account(acctno, birthday, balance)
Sales(saleid, sale_date, acctno, amount, status)
```

Concurrent access: table-level inconsistency

```
Update Sales S Set status = 'processing'
Where exists (Select * From Account A
              Where S.acctno = A.acctno AND A.balance > S.amount)
```

concurrent with …

```
Update Account Set balance = balance + 1000 where month(birthday) = 4;
```

Actions involved: Get, Modify, Put. They may be interleaved!

Account(<u>acctno</u>, birthday, balance)
Sales(<u>saleid</u>, sale_date, acctno, amount, status)

Concurrent access: multi-statement inconsistency

```
Insert Into Archive
  Select * From Sales Where status = 'paid';
Delete From Sales Where decision = 'paid';
```

concurrent with …

```
Select Count(*) From Sales;
Select Count(*) From Archive;
```

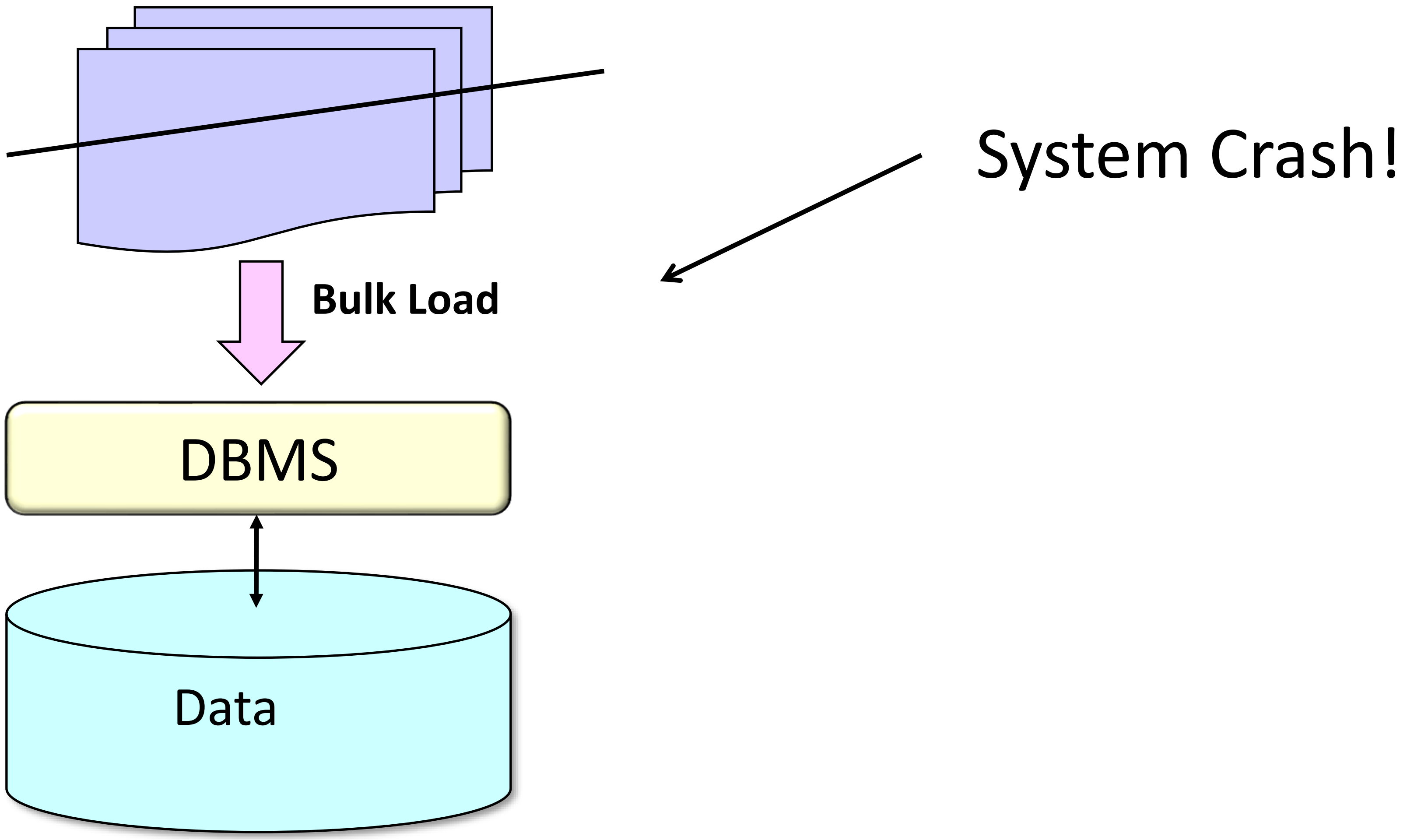
```
Account(acctno, birthday, balance)
Sales(saleid, sale_date, acctno, amount, status)
Archive(saleid, sale_data, acctno, amount, status)
```

Concurrency goal

- Execute sequence of SQL statements so that they appear to run in isolation
  - Simple solution?
    - Run them serially and in isolation.
    - But it's inefficient when they are accessing different objects.

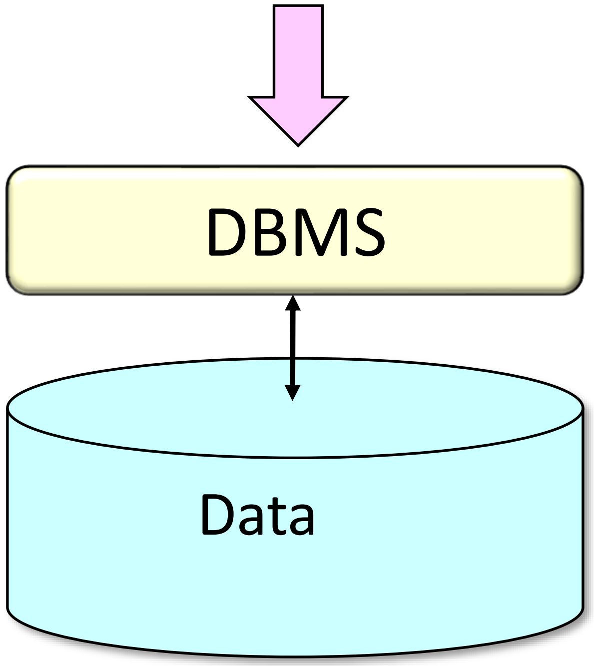
  - Need to enable concurrency whenever it is safe to do so.
    - Interleaving actions from two transactions to improve the overall performance

Motivation 2: resilience to system failures



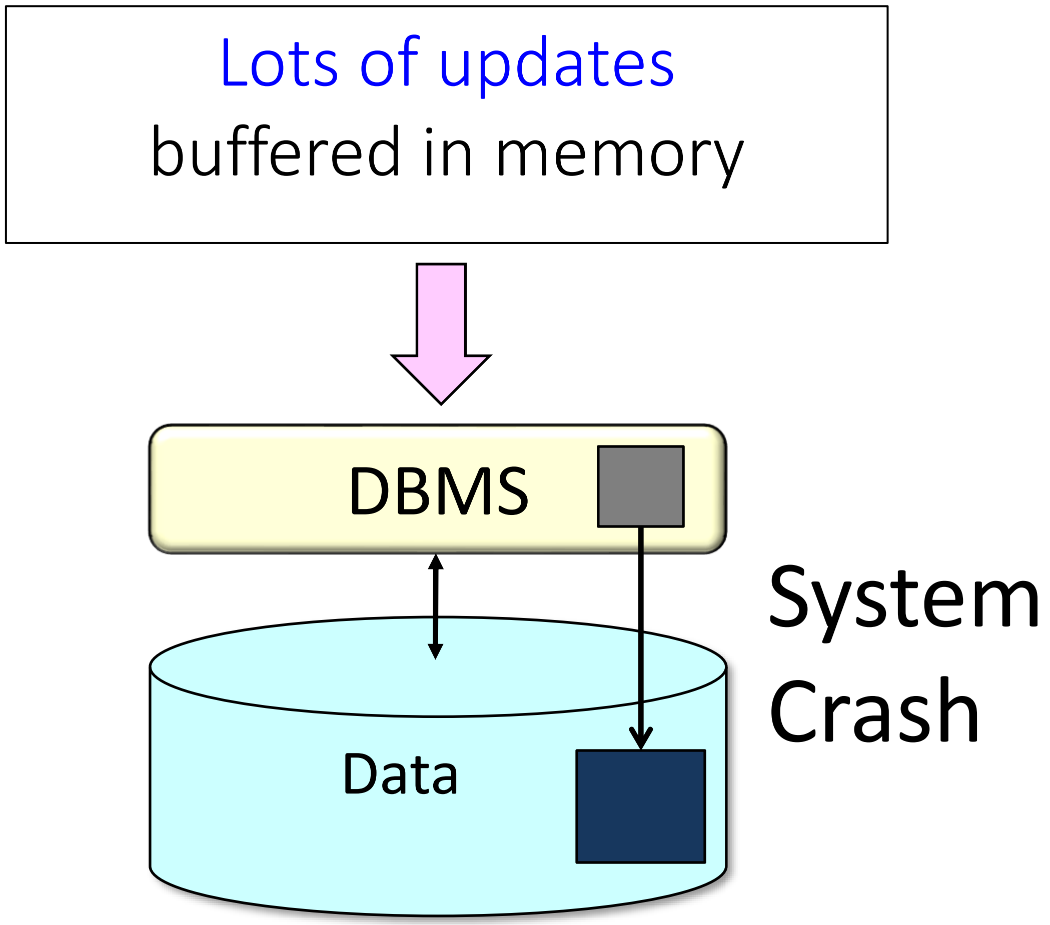
Example: system crash leads to data loss

```
Insert Into Archive
  Select * From Sales Where status = 'paid';
Delete From Sales Where status = 'paid';
```



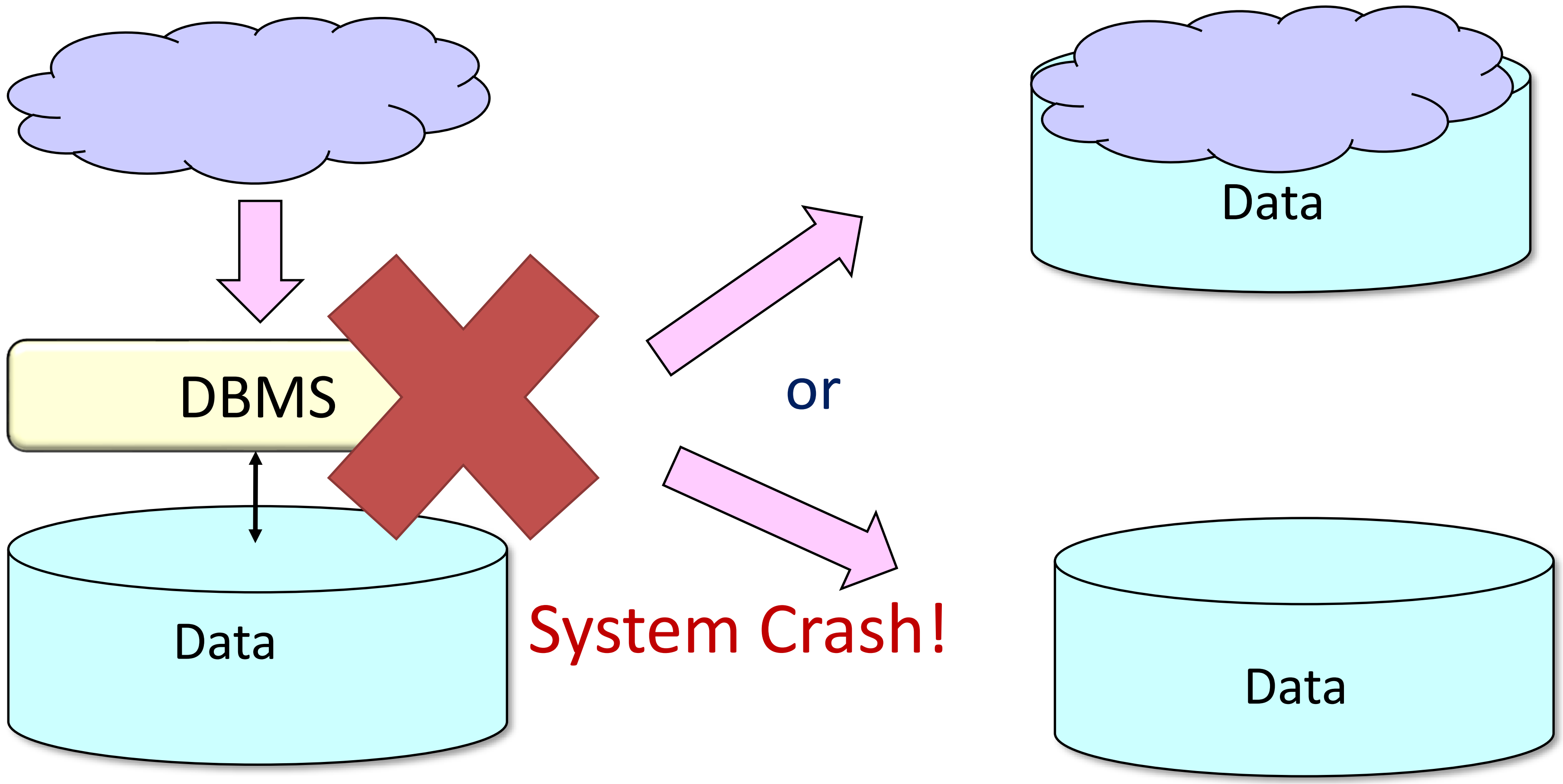
Crash failure

Example: system crash leads to data loss



System-failure goal

- Guarantee all-or-nothing execution, regardless of failures



Why using transaction?

Transaction:

```
BEGIN;
INSERT INTO A VALUES (…)
SELECT * from A;
DELETE FROM A WHERE …;
COMMIT;
```

- Transaction: a solution for both concurrency and failures
  - Transaction appear to run in isolation in the eye of the user
  - If the system fails, each transaction's changes appear in DB either entirely or not at all.

ACID Properties

- The desirable properties of transaction processing in DBMS.
  - Two important components
    - Concurrency control
    - Logging

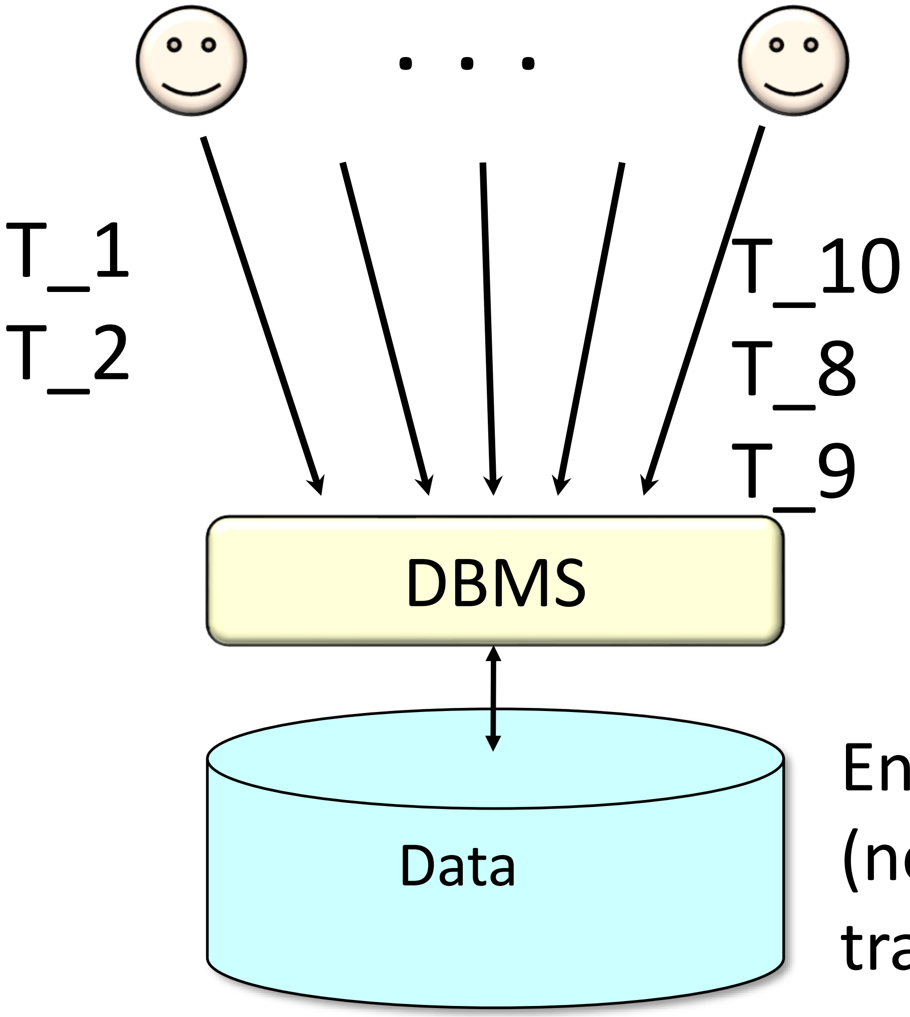
Atomicity

Consistency

Isolation

Durability

Isolation in ACID properties



Serializability

Operations may be interleaved, but execution must be equivalent to *some* sequential (serial) order of all transactions

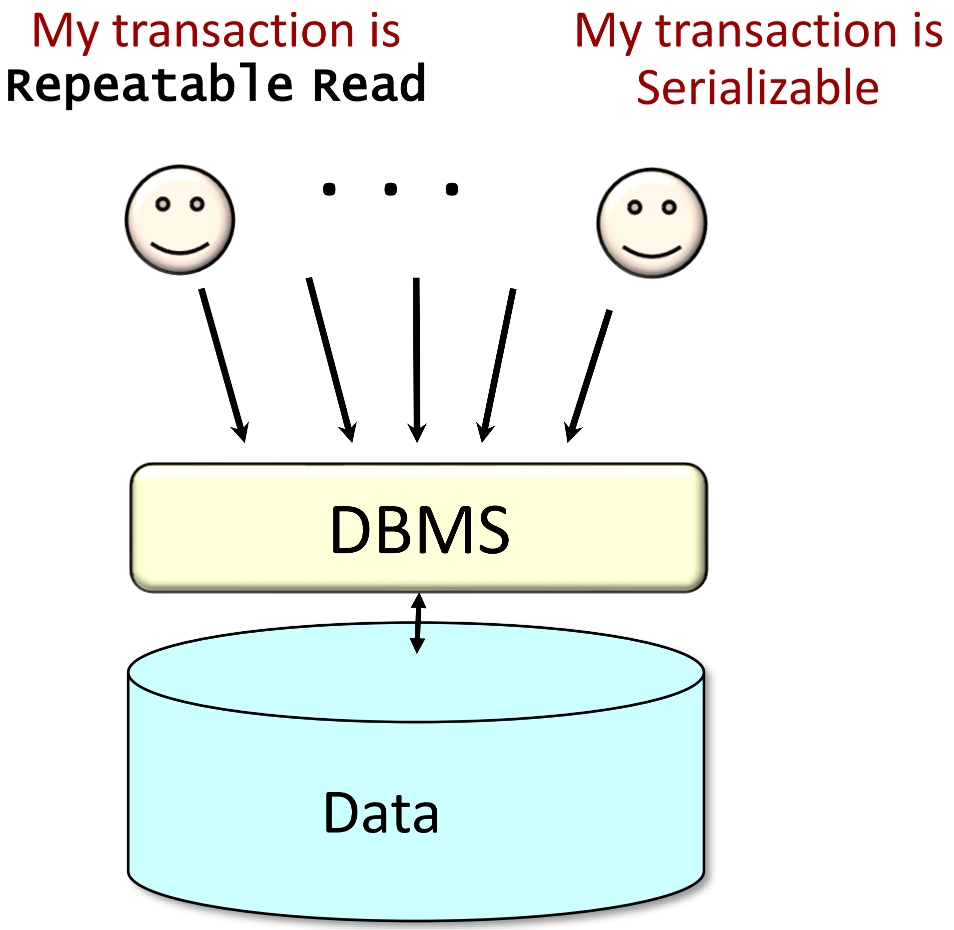
$\Rightarrow$ Overhead

$\Rightarrow$ Reduction in concurrency

End effect: T_1, T_10, T_8, T_9, T_2 (note that actions in different transactions may be interleaved!)

Isolation levels

- Isolation Levels
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE
- Per transaction
  - "In the eye of the beholder"
- All except serializable are defined by a few common anomalies
  - *Dirty read*
  - *Non-repeatable read*
  - *Phantom read*



Anomaly 1: Dirty read

- “Dirty” data item: written by an uncommitted transaction

```
Update Account Set balance = balance + 1000
Where month(birthday) = 4
```

concurrent with …

```
Select Avg(balance) From Account
```

- Dirty Reads: if read this value before the 1st Transaction has committed
- What happens if the 1st T rolls back after 2nd T has read this value?
  - non-serializable schedule

Anomaly 1: Dirty read

- "Dirty" data item: written by an uncommitted transaction

```
Update Account Set balance = 1.1 * balance Where month(birthday) = 4
```

concurrent with …

```
Select balance From Account Where acctno = 12345678
```

concurrent with …

```
Update Account Set birthday = DATE '1992-04-01' Where acctno = 12345678
```

Anomaly 2: Non-repeatable read

- Two reads to the same item emit different values in the same transaction.

Transaction 1

Transaction 2

```
Select balance From Account
Where acctno=12345678
```

```
Update Account Set balance= balance - 1000
Where acctno = 12345678;
COMMIT;
```

```
Select balance From Account
Where acctno = 12345678
```

Two reads in Xact 1 returns different values!
Note: it is allowed to return the value previously set in the same transaction.

Anomaly 3: phantom read

- A transaction
  - that might have avoided all dirty reads and non-repeatable reads
  - still does not guarantee serializability: because of the phantom read

Transaction 1

```
Select balance From Account
Where month(birthday) = 4
```

Transaction 2

```
INSERT INTO ACCOUNT VALUES
   (87654321, ‘1992-04-01’, 6000);
COMMIT;
```

```
Select balance From Account
Where month(birthday) = 4
```

Xact 1 queries the accounts whose owner were born in April twice. The second time includes something non-existent in the first time.

ANSI isolation levels

Anomalies / Isolatoin Level	Dirty Read	Non-repeatable read	Phantom Read
Read Uncommited	Possible	Possible	Possible
Read Committed	Impossible	Possible	Possible
Repeatable Read	Impossible	Impossible	Possible
Serializable *	Impossible	Impossible	Impossible

- *DBMS is allowed to provide stronger isolation level even if a weaker one is specified*
  - The table only describes the minimum requirements (i.e., the set of anomalies to prevent)
  - e.g., it is allowed to always provide serializable regardless of which isolation level is set
- *Serializable is defined by equivalence with serial schedule instead of anomalies!*
  - Same as free of the three anomalies with locking (2PL, discussed in next lecture)
  - Does cause issues for snapshot isolation (which admits additional anomalies, e.g., write skew)
    - Further reading: Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, Patrick E. O'Neil: A Critique of ANSI SQL Isolation Levels. SIGMOD Conference 1995: 1-10

READ ONLY transactions

- Helps system optimize performance
- Independent of isolation level

```
Set Transaction Read Only;
Set Transaction Isolation Level Repeatable Read;
Select Avg(balance) From Account;
Select Max(balance) From Account;
```

Isolation levels: summary

- Strongest isolation level: serializable
  - Worst performance but easiest to reason about
  - Note: serializable is often not the default isolation level in DBMS
    - for performance consideration
      - cause less performance surprise for novice users
      - looks better on benchmarks (if they are not careful)
      - but the implication is you have to be carefully reason about the program
        - or encounter weird bugs in production.
    - ***Takeaway: Always Read the Documentation of Transaction Behaviors!***
- Weaker isolation levels
  - Increased concurrency + decreased overhead = increased performance
  - Weaker consistency guarantees
  - Some systems have default Repeatable Read or even read committed
- Isolation level per transaction and “eye of the beholder”
  - Each transaction’s reads must conform to its isolation level

How to achieve ACID?

- An overview of ACID implementation

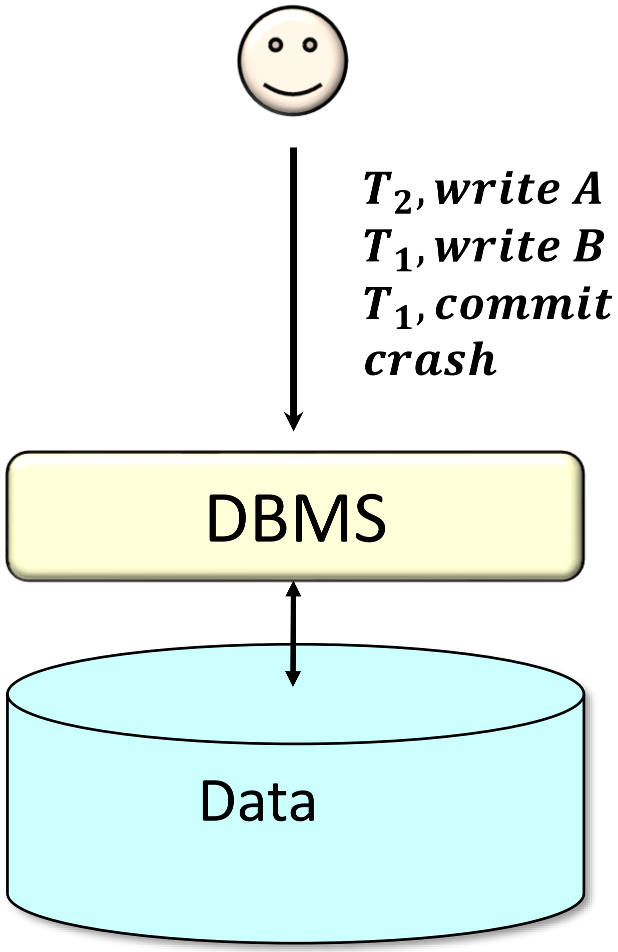
Atomicity

Consistency

Isolation

Durability

Atomicity in ACID properties



Each transaction is "all-or-nothing," never left half done

Achieved by Logging!

System needs to UNDO T2 in this case since it has NOT "Committed" at the time of crash.

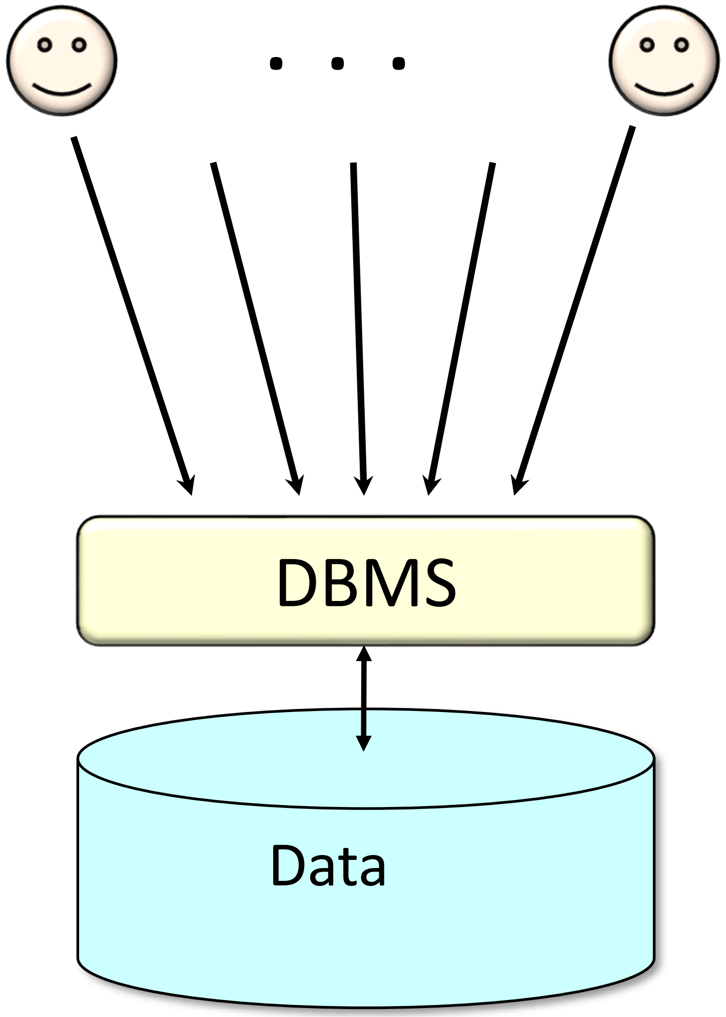
Transaction Abort

- Undoes partial effects of transaction
- Can be system or user initiated
  - System: crash recovery, serialization failure
  - User: calling ROLLBACK or ABORT, SQL errors (e.g., division by zero)

Each transaction is "all-or-nothing," never left half done

```
Begin Transaction;
<get input from user>
SQL commands based on input
<confirm results with user>
If ans='ok' Then Commit; Else Rollback;
```

Consistency in ACID properties

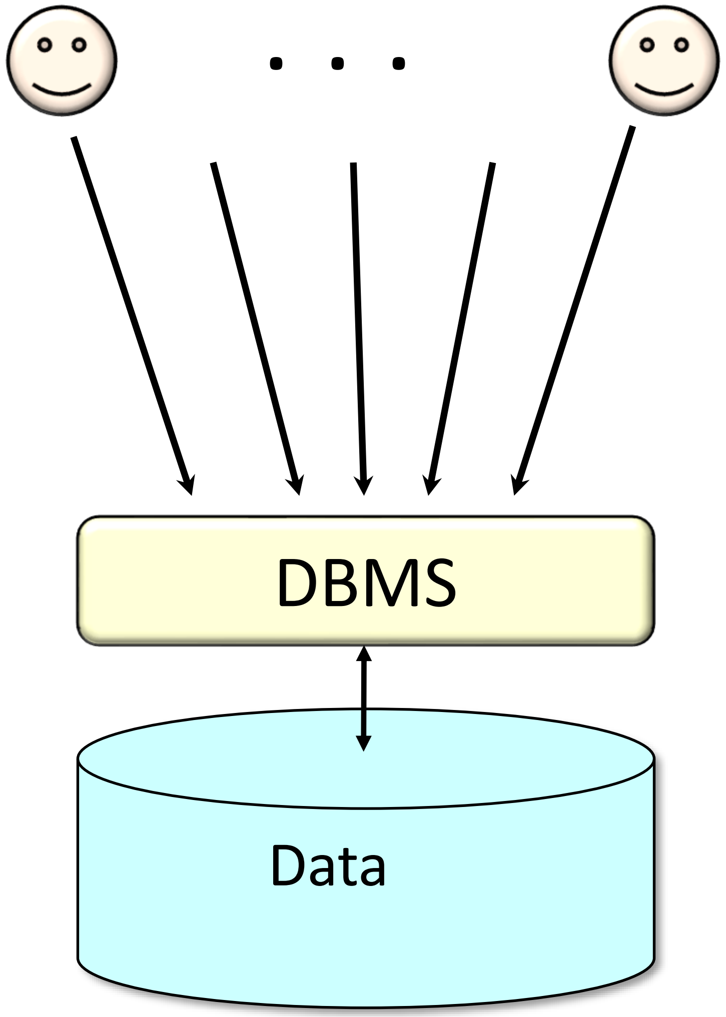


Each client, each transaction:

- Can assume all constraints hold when transaction begins
- Must guarantee all constraints hold when transaction ends

Serializability
+ Integrity constraint check for individual statements/transactions
⇒ constraints always hold

Isolation in ACID properties

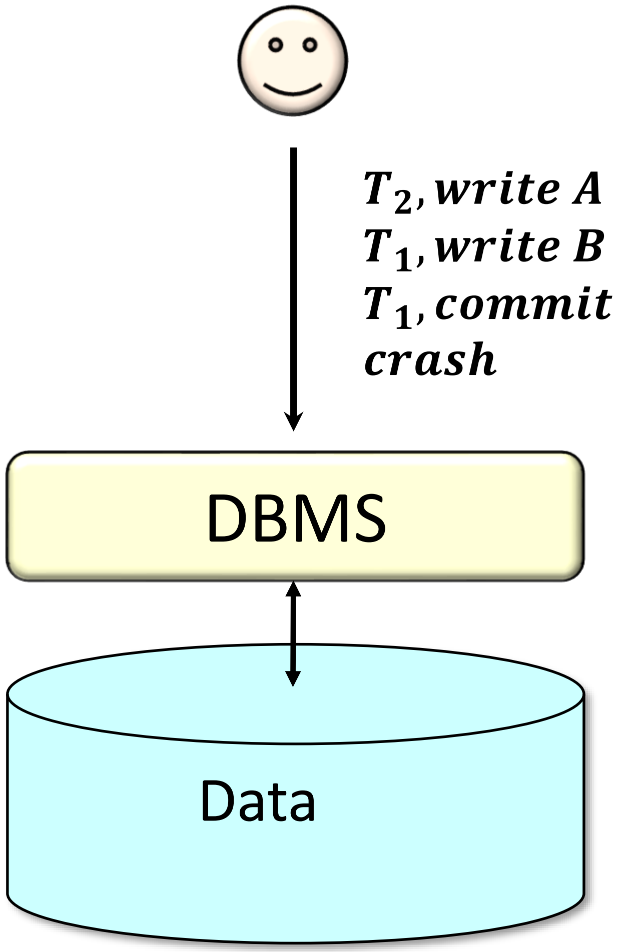


Serializability

Operations may be interleaved, but execution must be equivalent to *some* sequential (serial) order of all transactions

Achieved by Concurrency Control!
e.g., Locking.

Durability in ACID properties



If system crashes
after transaction commits,
all effects of transaction
remain in database

Achieved by Logging!

System may need to REDO T1 in this case since it has "Committed".

Summary

- This lecture
  - Transaction
  - Isolation level
  - ACID properties

- Next lecture
  - Pessimistic Concurrency Control (i.e., locking)